Procedures

Procedures are

- subroutines
- functions

Procedures are

- module
- internal
- external

All Fortran 77 procedures are external. It is easy to convert external procedures to module procedures, which are much easier to use. Next slide

The `contains` Statement

The `contains` statement marks the beginning of internal or module procedures, which must be placed just before the `end` statement.

A module procedure may contain internal procedures, but internal procedures may not contain internal procedures. Previous slide Next slide

Case Study: Sorting

```
program sort_3

   implicit none
   real :: n1, n2, n3

   call read_the_numbers
   call sort_the_numbers
   call print_the_numbers
```

Previous slide Next slide

```
contains

subroutine read_the_numbers
   read *, n1, n2, n3
   print *, "Input data  n1:", n1
   print *, "            n2:", n2
   print *, "            n3:", n3
end subroutine read_the_numbers
```

Previous slide Next slide

```
subroutine sort_the_numbers
   if (n1 > n2) then
      call swap (n1, n2)
   end if
   if (n1 > n3) then
      call swap (n1, n3)
   end if
   if (n2 > n3) then
      call swap (n2, n3)
   end if
end subroutine sort_the_numbers
```

Previous slide Next slide

```
subroutine print_the_numbers
   print *,  &
   "The numbers, in ascending order, are:"
   print *, n1, n2, n3
end subroutine print_the_numbers

subroutine swap (a, b)
   real :: a, b, temp
   temp = a
   a = b
   b = temp
end subroutine swap

end program sort_3
```

Learn more about internal procedures. Previous slide Next slide

Host Association

In the subroutines `read_the_numbers`, `sort_the_numbers`, and `print_the_numbers`, the variables `n1`, `n2`, and `n3` are known from the main program by *host association*.

Learn more about host association. Previous slide Next slide

Local Variables

In the subroutine `swap`, the variable `temp` is *local*. The dummy arguments `a` and `b` are also local. If any of these names were to be used outside the internal procedure `swap`, they would be different entities.

Learn more about scope. Previous slide Next slide

Exercises

1. Write parameter statements to set values for a loan principal amount *P* of $106,500, an annual interest rate *R* of 7.25%, and the number of months *M* equal to 240 in which the loan is to be paid off. Write a subroutine or function PAY to compute the monthly payment given by the formula:

   $$PAY = [r \times P\ (1+r)^M] / [(1+r)^M - 1]$$

   where *r* is the monthly interest. Note that if the annual interest rate is *R*, the monthly interest rate *r* is R/12. Write a program that tests PAY as an internal subroutine.
2. *Write another subroutine to print out a monthly schedule of the interest, principal paid, and remaining balance.

Previous slide Next slide

Function Result

Within a function, the result value of the function can be held in a variable whose name is different from the function name. This is necessary when the function calls itself directly, but may avoid confusion in other cases. The next example computes

$f(x) = (1 + 1/x)^x$

for values of x equal to 1, 10, 100, ..., 10^{10} Previous slide Next slide

```
program function_values

   implicit none
   real :: x
   integer :: i

   do i = 0, 10
      x = 10.0 ** i
      print "(f15.1, f15.5)", x, f (x)
   end do
```

Previous slide Next slide

```
contains

function f (x)  result (f_result)

   real :: f_result, x
   integer, parameter :: &
      kind_11 = selected_real_kind (11)

   f_result =  &
      (1 + real (1 / x, kind_11)) ** x

end function f

end program function_values
```

Learn more about functions.
Learn more about subroutines. Previous slide Next slide

Agreement of Arguments

Except for optional arguments the number of actual and dummy arguments must be the same. Except when keywords are used, the order of the arguments must be the same. The corresponding actual and dummy arguments should agree in

- Type
- Kind
- Shape
- Length (if character)

These rules are a little more restrictive than required, but observing them eliminates no functionality. The rules are easy to follow when assumed-shape and assumed-length dummy arguments are used.

Learn more about passing arguments. Previous slide Next slide

Argument Intent

The intended use of a dummy argument can be indicated with the `intent` attribute. This can provide some additional compiler checks and aid optimization.

```
intent (in)        Procedure must not change it
intent (out)       Must be definable;
                      Dummy undefined on entry
intent (inout)     Must be definable
```

Declare the intent for all procedure arguments.

Learn more about argument intent. Previous slide Next slide

Keyword Arguments

A procedure may be called using keyword arguments. When a keyword argument is used or an optional argument is omitted, all arguments after that one must use keywords, but they may come in any order. Suppose a subroutine `s` has four real arguments `a`, `b`, `c`, and `d`. Then any of the following are equivalent legal calls to `s`.

```
call s (w, x, y, z)
call s (w, x, y, d=z)
call s (w, x, d=z, c=y)
call s (d=z, b=x, c=y, a=w)
```

Note: for external procedures, you need an interface block to do this.

Learn more about argument keywords. Previous slide Next slide

Optional Arguments

A procedure may declare that some arguments are optional. The `present` intrinsic function is used in the procedure to determine if the argument has been passed. Note: for external procedures, you need an interface block to do this.

A rule: In a procedure, an argument that is optional must not present must not be referenced, except to pass it along to another procedure. The intrinsic function `present` is used to determine whether or not an argument is present.

Another rule: Going from left to right, if an actual argument is omitted or is passed using a keyword, all remaining arguments must be passed using a keyword. Previous slide Next slide

```
function sum_numbers (from, to)

   integer, intent (in), optional :: from
   integer, intent (in) :: to
   integer :: sum_numbers

   if (present (from)) then
      sum_numbers = (to+from)*(to-from+1)/2
   else
      sum_numbers = to*(to+1)/2
   end if

end function sum_numbers
```

Previous slide Next slide

This function can be called with the first argument "missing".

```
program test_sum

   implicit none

   print *, sum_numbers (2, 10)
   print *, sum_numbers (to=10, from=1)
   print *, sum_numbers (to=10)

contains

   function sum_numbers (from, to)
      . . .
   end function sum_numbers

end program test_sum
```

Learn more about optional arguments. Previous slide Next slide

```
program integrate

implicit none

intrinsic sin

print *, integral  &
   (sin, a=0.0, b=3.14159, n=100)
print *, integral (sin, a=0.0, b=3.14159)
print *, integral (b=3.14159, a=0.0, f=sin)
print *, integral (b=3.14159, f=sin)

contains
```

Previous slide Next slide

```
function integral (f, a, b, n)  &
      result (integral_result)
!  Calculates a trapezoidal approximation
!  to an area using n trapezoids.

!  The region is bounded by lines x = a,
!  y = 0, x = b, and the curve y = f (x).

   real :: integral_result, f
   real, intent (in), optional :: a
   real, intent (in) :: b
   integer, intent (in), optional :: n
   real :: h, sum, aa
   integer :: i, nn
```

Previous slide Next slide

```
if (present (a)) then
   aa = a
else
   aa = 0.0
end if

if (present (n)) then
   nn = n
else
   nn = 100
end if
```

Previous slide Next slide

```
   h = (b - aa) / nn
!  Calculate the sum f(a)/2+f(a+h)+...
!                   +f(b-h)+f(b)/2
!  Do the first and last terms first
   sum = 0.5 * (f (aa) + f (b))
   do i = 1, nn - 1
      sum = sum + f (aa + i * h)
   end do

   integral_result = h * sum

end function integral

end program integrate
```

Previous slide Next slide

Interface Blocks

If the function `sum_numbers` is to be an external function, the calling program must contain an interface block to describe the arguments and the type of the result.

```
interface
function sum_numbers (from, to)
   integer, intent (in), optional :: from
   integer, intent (in) :: to
   integer :: sum_numbers
end function sum_numbers
end interface
```

Previous slide Next slide

An interface block is also used to describe a dummy argument that is a procedure. Thus, `f` in the function `integral` should be declared:

```
interface
   function f (x) result (f_result)
      implicit none
      real, intent (in) :: x
      real :: f_result
   end function f
end interface
```

Previous slide Next slide

If the function `integral` is to be an external function, the calling program must contain an interface block for it.

```
interface
function integral (f, a, b, n)  &
      result (integral_result)
   implicit none
   real :: integral_result
   interface
      function f (x) result (f_result)
         implicit none
         real, intent (in) :: x
         real :: f_result
      end function f
   end interface
   real, intent (in), optional :: a
   real, intent (in) :: b
   integer, intent (in), optional :: n
end function integral
end interface
```

Learn more about interface blocks. Previous slide Next slide

The following ten situations require explicit interfaces for external procedures:

- optional arguments
- array-valued functions
- pointer-valued functions
- character-valued functions whose lengths are determined dynamically
- assumed-shape dummy arguments (needed for efficient passing of array sections)

Previous slide Next slide

- dummy arguments with the pointer or target attribute
- keyword actual arguments (which allow for better argument identification and order independence of the argument list)
- generic procedures (calling different procedures with the same name)
- user-defined operators (which is just an alternate form for calling certain functions)
- user-defined assignment (which is just an alternate form for calling certain subroutines)

Previous slide Next slide

Exercise

1. Modify the loan calculation subroutine so that it has three dummy arguments `R`, `P`, and `M` and the number of months `M` is optional and is 240 if not present. Test it with

   ```
   call pay_calc (R=7.25, P=106500.00)
   ```

   Previous slide Next slide

Pure Procedures (F95)

In order for computational results to be determinate, functions must not have side effects such as changing values in common or writing intermediate results to a file. Pure procedures are intended to disallow the side effects that impact determinancy. User-defined pure procedures may be used in specification expressions.

A function is declared pure by putting the keyword `pure` on the `function` statement.

```
pure function f(x) result(the_answer)
```

Learn more about pure procedures. Previous slide Next slide

All of the intrinsic functions and the `mvbits` intrinsic subroutine are pure. The prefix specification `pure` in a user-defined function or subroutine statement specifies that procedure to be pure. Previous slide Next slide

There are four contexts in which a procedure must be pure:

- a function referenced in a `forall` construct
- a function referenced in a specification statement
- a procedure that is passed as an actual argument to a pure procedure
- a procedure referenced in the body of a pure procedure (including those referenced by a defined operator or defined assignment)

Previous slide Next slide

Elemental Procedures (F95)

The purpose of elemental procedures is to allow the programmer to define a procedure with scalar arguments and the elemental keyword that can be called with array arguments of any rank. An elemental procedure has all scalar dummy arguments; in addition, an elemental function delivers a scalar result. The actual arguments may be arrays of any rank, as long as all of the actual arguments in a given call are in general conformable. The result of an elemental call having array actual arguments is the same as would have been obtained if the procedure had been applied separately, in any order (including simultaneously), to the corresponding elements of each argument. The prefix specification `elemental` in a user-defined function or subroutine statement specifies that procedure to be elemental. Previous slide
Next slide

```
elemental function vip_calc(x, y)
   real :: vip_calc
   real, intent(in) :: x, y
      . . .
end function vip_calc

!  A call to vip_calc with scalar arguments
   x = vip_calc(1.1, 2.2)

! The result of this call is an array
!    conformable with a(1:n) and b(1:n).
  ax = vip_calc(a(1:n), b(1:n))
```

Learn more about elemental procedures. Previous slide Next slide

Scope

The scope of a name is the set of lines in a Fortran program where that name may be used and refer to the same variable, procedure, or type.

The scope of a variable or type name declared in a main program, module, module procedure, or external procedure extends from the first statement to the `end` statement or to the `contains` statement, if there is one.

Learn more about scope. Previous slide Next slide

Any name that is declared is known by *host association* in all procedures contained in the one in which it is declared, that is, in all procedures following the `contains` statement. However, the name declared is not known in any internal or module procedure in which the name is redeclared.

A name declared in an internal procedure has scope extending only from the beginning to the end of that procedure, not to the program or procedure that contains it, nor to any other internal procedure. Previous slide Next slide

```
program p
   implicit none
   integer :: a, b
   . . .
contains

subroutine s
   real b
   . . .
   print *, a, b
   . . .
```

Previous slide Next slide

In the subroutine, `a` is the integer variable declared in the main program; it is known in the subroutine by host association because it is not redeclared. However, it is a real value that is printed for `b`, which is the `b` declared in the subroutine `s`.

A similar rule applies to `implicit` statements. The implicit typing (including `none`) is passed along to any internal or module procedures.

The name of an internal procedure, the necessary information about its arguments, as well as the type of its result variable if it is a function, are considered as declared in the containing program or procedure, and so they are known throughout the containing program and all other internal procedures of the containing program. The containing program or procedure therefore can call an internal procedure, as can any other internal procedure of the same containing procedure. Previous slide Next slide

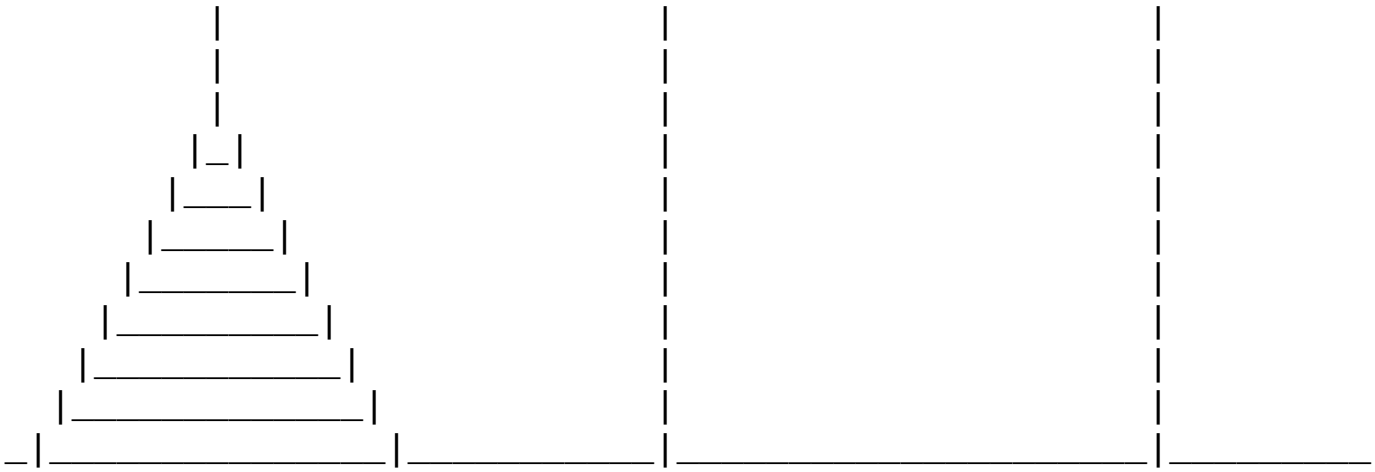
The save Attribute

Variables local to a procedure do not retain their values when the procedure is exited unless they have the `save` attribute. A variable that is initialized is also saved, but it is a good idea to give something the `save` attribute explicitly when it is to be saved.
The variable `counts` records the number of times the subroutine is called:

```
subroutine subr ()
   integer, save :: counts = 0
   . . .
   counts = counts + 1
   . . .
end subroutine subr
```

Learn more about the `save` attribute. Previous slide Next slide

Recursion



Problem: move the disks from the left post to the right post, obeying the rules:

1. Disks must be moved from post to post one at a time.
2. A larger disk may never rest on top of a smaller disk on the same post.

Previous slide Next slide

The algorithm has 3 steps.

1. Legally move the top *n*-1 disks from the starting post to the free post.
2. Move the largest disk from the starting post to the final post.
3. Legally move the *n*-1 disks from the free post to the final post.

Previous slide Next slide

```
program test_hanoi

   implicit none
   integer :: number_of_disks

   read *, number_of_disks
   print *, "Input data  number_of_disks:",  &
         number_of_disks
   print *
   call hanoi (number_of_disks, 1, 3)

contains
```

Previous slide Next slide

```
recursive subroutine hanoi (number_of_disks,  &
      starting_post, goal_post)

   integer, intent (in) ::  &
   number_of_disks, starting_post, goal_post
   integer :: free_post
   integer, parameter :: all_posts = 6

   if (number_of_disks > 0) then
      free_post =  &
      all_posts - starting_post - goal_post
      call hanoi (number_of_disks - 1,  &
                  starting_post, free_post)

      print *, "Move disk", number_of_disks,  &
            "from post", starting_post,  &
            "to post", goal_post
      call hanoi (number_of_disks - 1,  &
                  free_post, goal_post)
   end if
end subroutine hanoi

end program test_hanoi
```

Learn more about recursion. Previous slide Next slide

```
run test_hanoi

 Input data  number_of_disks: 4

 Move disk 1 from post 1 to post 2
 Move disk 2 from post 1 to post 3
 Move disk 1 from post 2 to post 3
 Move disk 3 from post 1 to post 2
 Move disk 1 from post 3 to post 1
 Move disk 2 from post 3 to post 2
 Move disk 1 from post 1 to post 2
 Move disk 4 from post 1 to post 3
 Move disk 1 from post 2 to post 3
 Move disk 2 from post 2 to post 1
 Move disk 1 from post 3 to post 1
 Move disk 3 from post 2 to post 3
 Move disk 1 from post 1 to post 2
 Move disk 2 from post 1 to post 3
 Move disk 1 from post 2 to post 3
```

Previous slide Next slide

`include` Line

An `include` line contains the keyword `include` and a character literal constant. The meaning of the constant is not specified, but probably is a file name on most systems. The `include` line is replaced by included text, such as the contents of a specified file.

```
include "my_definitions.inc"
```

Although the `include` line might be useful in some simple cases, the use of modules is recommended instead.

Learn more about the `include` line. Previous slide Next slide

Exercise

1. Write a recursive function `bc (n, k)` to compute the binomial coefficient

$$\binom{n}{k}$$

Previous slide